Airflow Backfill Design

Zhuo (Joe) Wang Sanitized Version
Created on 02/13/2023
Reviewed on 02/14/2023
Reviewed on 02/27/2023
Modified for open source on 10/29/2023

Terms

- **Backfill:** Backfill is a process to run a DAG for a specific date range in the past. The typical use cases which requires backfill of a DAG are:
    - Rerun a DAG for affected dates after upstream data is changed
    - Rerun a DAG for all the dates after the DAG definition is changed
    - Data got corrupted for a specific date range
- **Origin DAG**: the DAG to be backfilled
- **Scheduled Runs**: the current or future DAG runs of an origin DAG

Problem Statement

Existing Backfill Solutions in Ariflow

After diving into Airflow (version 2.3.2), we found solutions available in the Airflow that have gaps from our use cases.

Trigger DAG run from UI
Users can trigger a DAG run for a logical date from the UI directly, but only for one day at a time. This can serve as an ad hoc backfill of a few dates, but won't work for a large batch. Another issue is that DAG runs, triggered by this way, share the same resource limit like maximum parallel runs with the origin DAG, which may block scheduled runs.

Catchup
Catchup is used to trigger missing DAG runs for the entire history from start date to end date (usually to current). It can "backfill" if properly removing DAG runs of to-be-backfilled dates. This causes problem when:

- Catchup can be enabled for a DAG from the DAG definition itself. If Catchup was not enabled originally, after enabling for backfill purpose, a Catchup will start to fill in all missing dates, which may not be what users want
- Catchup runs share the resource limit of scheduled runs, so catchup runs may block scheduled runs.

Airflow CLI

Airflow has built-in Backfill CLI support, it can run backfill from command line, but it has some issues:

- Not maintainable or scalable
  Backfill CLI can run internally within the Airflow scheduler and use its resources. This impacts the schedulers, and makes the schedulers hard to maintain. This is also hard to scale up with the backfill requests due to limited scheduler resources.
- Not manageable
  Backfills triggered through CLI are not manageable through UI or CLI as separate entities. Users can not see the status of the backfill or resume any failed backfill.

Other Solutions
We did research and found a few other solutions before implementing our own, they all use Airflow CLI as the base implementation therefore suffer the similar issues with Airflow CLI. Please refer to the Appendix for details.

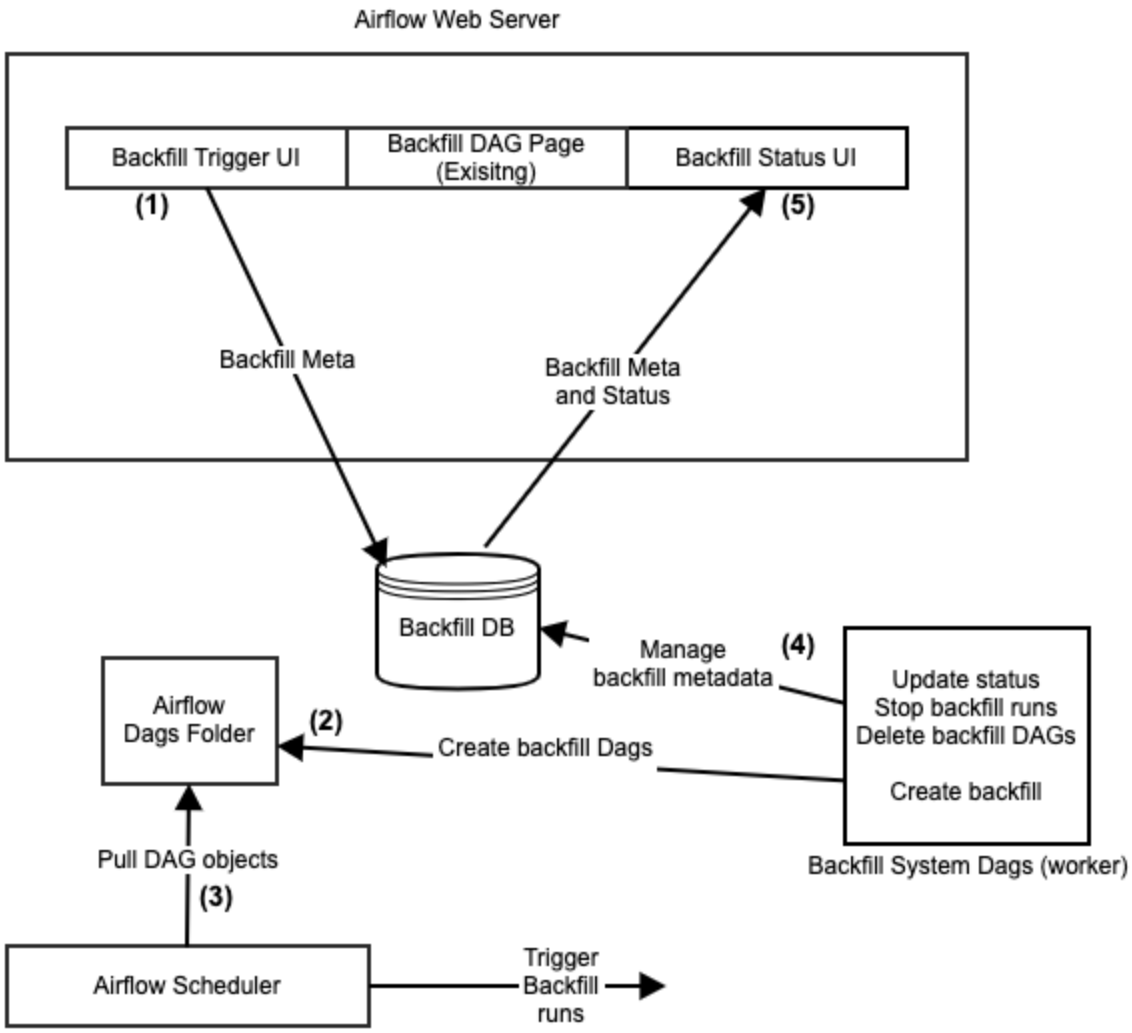
Key Requirements

The Backfill solution should be manageable, scalable and easy to use. Here are the key requirements for it:

- Users can trigger backfills, track status from UI
- Backfills run independently, not blocking scheduled DAG runs
- Backfills can recover automatically after Airflow recovers
- Resources used by backfills can be controlled

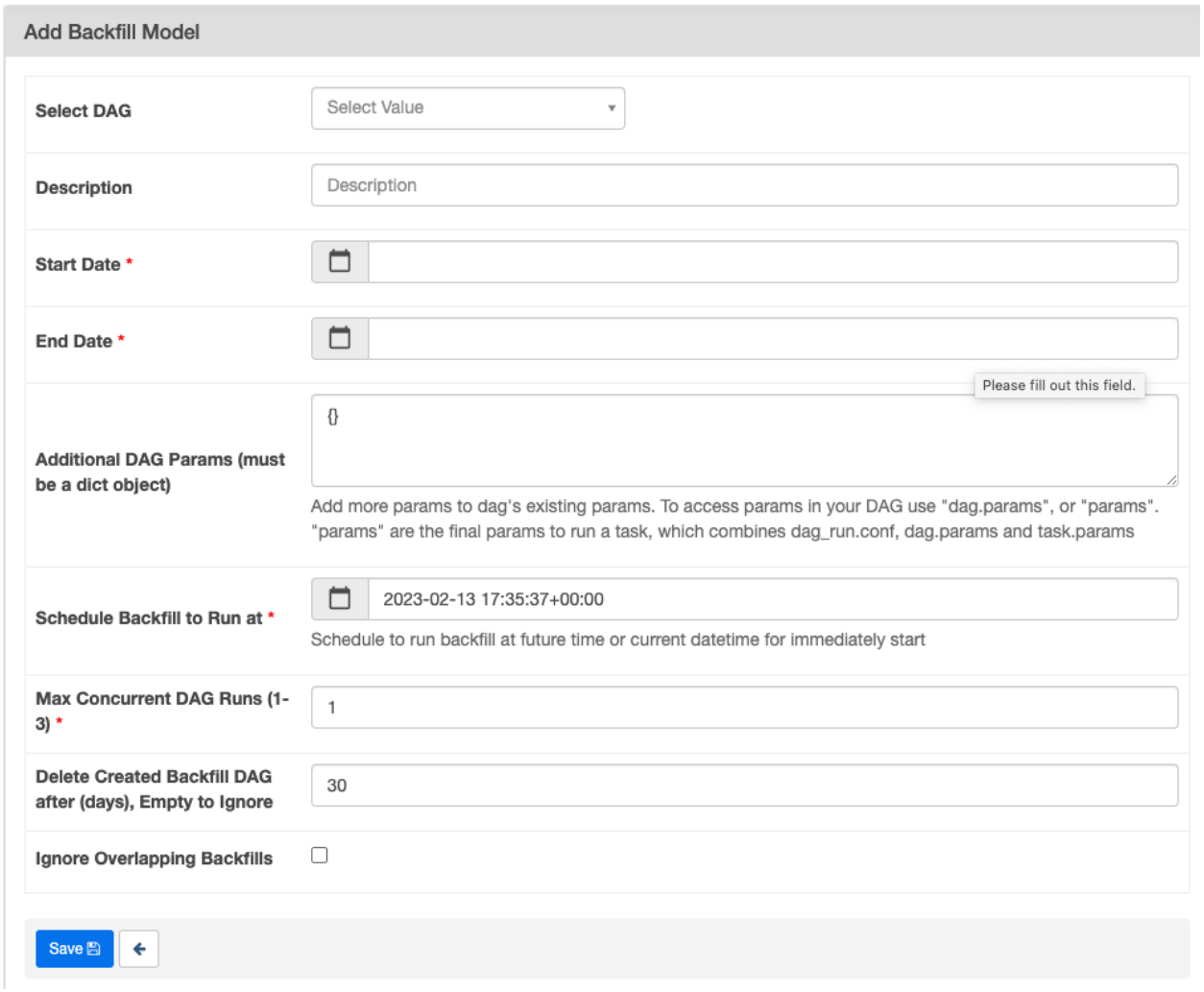
Design

The basic idea is to create a separate DAG for backfill purpose dynamically, Airflow scheduler then takes over to run the backfill DAG.
The following graph is the structure of the backfill. The steps are marked with numbers in the graph.



1. The backfill process starts with the Backfill Trigger UI. The backfill permissions are enforced at submit by Airflow's RBAC according to user permissions of can_edit. The following image is the backfill trigger UI. Users can provide origin DAG, date range, run params, and other related parameters to trigger a backfill.

   When submitting, the backfill meta of origin Dag Id, backfill start date, backfill end date, max active runs, and other related info is persisted into a database table (backfill table, see Appendix for table structure).



2. Backfill creation Dag runs periodically. When a new backfill request was detected, a backfill Dag file is created in the Airflow Dags folder according to the backfill meta and origin Dag source code.

   The backfill Dag id is generated according to the customizable naming conventions, which by default adding “backfill” and timestamp affix.

   The backfill Dag has “Catchup” enabled with requested start date and end date.

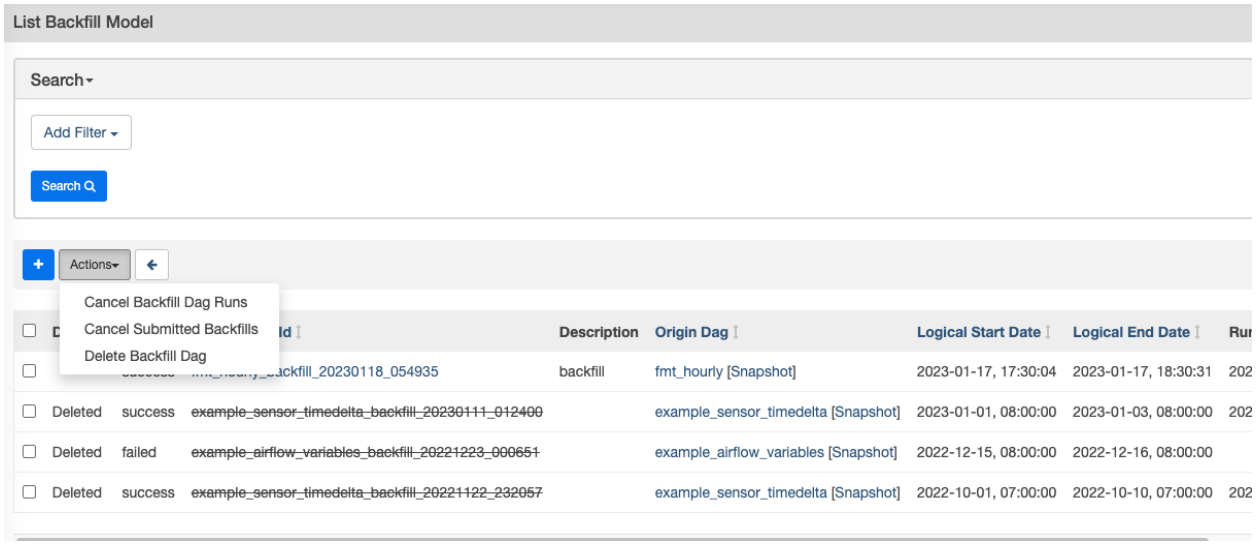
3. When the Airflow scheduler periodically loads dags from the dag folder, backfill Dags are loaded as other regular Dags and get scheduled accordingly.

   The backfill Dags will automatically start to run without interrupting the runs of origin Dags, and the backfill Dag runs are controlled by the dag-level max_active_run to limit resource usage. Users can get the status and logs of backfill Dag runs from the Dag page UI.

4. Backfill system Dags (run at workers) will periodically update the backfill status to the backfill table according to the backfill Dag run status.

5. Backfill Management UI pulls the backfill meta, the status from the database and shows them to users. Users can also manage backfills, under the hood, through web server lib function calls to:
    - Cancel backfill runs
    - Delete backfill Dags (They can't be deleted from Dags UI because backfill Dag files are not deleted)

    Here is the the Backfill Management UI:



Appendix

Other Airflow Backfill Solutions

1. [miliar/airflow-backfill-plugin]
   This is a UI wrapper of Airflow Backfill CLI, it runs CLI at Web servers. This does not work anymore after Airflow 2 when Web Server does not store DAG code anymore.

2. [AnkurChoraywal/airflow-backfill-util]
   This is another UI wrapper of Airflow Backfill CLI, and suffers the same problem with the solution above.

3. [Backfill Queue and Backfill on Demand]
   These only appeared in discussion, no implementation found.
   Backfill Queue: Redis is used to queue backfill requests, and workers run Airflow Backfill CLI according to queue.

Backfill on Demand: chatops server is used to spin up K8s containers to run Airflow Backfill CLI.

These solutions solved scalable issues by spinning up workers to run Airflow CLI, but they still suffer from manageable issues due to Airflow CLI. Additionally these solutions need more investments on implementing a “backfill scheduler” to control backfill runs. Please refer to the origin discussion for the details of pros and cons.

Backfill Table Definition

```
class BackfillModel(Base, LoggingMixin):
    """Table containing backfill properties"""

    __tablename__ = "custom_backfill"    # add custom prefix for non origin airflow tables

    # backfill dag id
    backfill_dag_id = Column(String(ID_LEN), primary_key=True)

    # id of the dag to be backfilled
    dag_id = Column(String(ID_LEN), nullable=False)
    # state
    state = Column(Enum(BackfillState), nullable=False)
    # user description of the backfill
    description = Column(VARCHAR(1000))
    # backfill start date
    start_date = Column(UtcDateTime, nullable=False)
    # backfill end date
    end_date = Column(UtcDateTime, nullable=False)
    # logical dates included in the range
    logical_dates = Column(Text)
    # dag params
    dag_params = Column(Text)
    # max active runs
    max_active_runs = Column(Integer, nullable=False)
    # backfill scheduled date
    scheduled_date = Column(UtcDateTime)
    # delete date
    delete_date = Column(UtcDateTime)
    # origin dag file relative location
    relative_fileloc = Column(VARCHAR(1000))
    # origin code snapshot
    origin_code = Column(Text)
    # submit date
    submitted_at = Column(UtcDateTime)
    # submit user
    submitted_by = Column(VARCHAR(50))
    # update date
    updated_at = Column(UtcDateTime)
    # deleted
    deleted = Column(Boolean, default=False)
    # backfill run start date
    run_start_date = Column(UtcDateTime)
    # backfill run end date
    run_end_date = Column(UtcDateTime)
```